Pascal NEWSLETTER

NUMBER 7 OREGON SOFTWARE SUMMER/FALL 1983

UNIX compiler for 68000 released

At the USENIX Conference in Toronto, Oregon Software introduced the first high-performance Pascal compiler for the growing UNIX market on the Motorola MC68000. The compiler is available by itself or with the source-level, interactive Debugger and other software development utilities.

UNIX, developed by Bell Labs and licensed through Western Electric, is becoming a major development system on the 68000. Several dozen vendors are now offering UNIX or UNIX-like systems for applications development on the MC68000. Pascal–2 offers the benefits of Pascal without sacrificing the code quality required in the UNIX production environment.

Like other Pascal–2 products, the UNIX compiler performs eight optimizations on user programs, producing small, fast code. Oregon Software's benchmarks show that code produced by the Pascal–2 compiler is more compact and efficient than that of other high-level languages on 68000 UNIX systems, including C, and is an order of magnitude faster than interpretive or threaded languages.

Debugger improved

A key feature of UNIX is the concept of supporting "tools" — standard, easy-to-use programs that aid a programmer in coding and development. Like Oregon Software's existing Pascal packages, Pascal–2 for UNIX contains a set of development tools — a debugger, execution profiler, program and text formatters, and cross referencers for program analysis.

As with the other Pascal–2 debuggers, the UNIX debugger allows the programmer to interactively solve logic errors in applications at the level of the source program, thus simplifying and speeding development. The UNIX debugger, however, runs as a separate process from the application code being debugged, keeps track of multiple compilation units, and performs breakpoint debugging without slowing down the application code.

The profiler and other utilities perform substantially the same as they do on other systems.

Product unbundled

Pascal–2 for UNIX may be purchased as a complete package. Users also have the choice of purchasing the compiler only, and may opt for short-term or annual support.

Documentation includes the *Pascal–2 User Manual* with User's and Programmer's Guides, Debugger and Utilities Guides, and a detailed Language Specification; inserts for the UNIX Programmer's Reference Manual; and two textbooks on programming in Pascal.

Version 2.1B release nears

Pascal–2 Version 2.1B is currently in field test. Release to general users is scheduled for Nov. 1. Numerous V2.1A bugs have been fixed and the ability to debug external procedures has been added. All supported users may request updated software and documentation from Oregon Software or their distributor.

In this issue...

UNIX/68000 compiler released . . . . . Page 1
RSTS SourceTools available . . . . . Page 2
Pascal–2 on DEC Pro 350 . . . . . Page 2
New releases for VAX, VERSAdos . . . . . Page 3
V2.1B ready . . . . . Page 4
Using unformatted files . . . . . Page 4
Letters . . . . . Page 6
OPUS Communiqué . . . . . Page 7
Accessing global symbols . . . . . Page 7
Pascal–2's concurrency package . . . . . Page 8
Pascal–1 real-time facilities . . . . . Page 11
The Log . . . . . Page 18
Errors, additions to manuals . . . . . Page 20
Information exchange . . . . . Page 21
Pascal standard . . . . . Page 21
Sales staff . . . . . Page 22
Contest winners . . . . . Page 22
Transition marked . . . . . Page 23

Source Tools also available for RSTS/E

Oregon Software's SourceTools, a software management system, is now available for the RSTS/E operating system. The product was earlier released for the VAX/VMS and RSX-11M operating systems.

SourceTools allows programmers to manage access to sources and to document changes in sources over the development life cycle of a product.

In the past, software developers have relied on informal, manual methods to manage software development. With software projects growing in complexity and development teams increasing in size, these traditional methods of project management have eroded the real productivity of programmers.

SourceTools addresses this problem by providing an effective, automatic management scheme for software projects, particularly those involving numerous source files, joint development by several programmers, or the creation of cross-system software. SourceTools may be used with software developed in any computer language and with any standard text file.

The core of SourceTools is a package called SourceCon that controls the creation and modification of source files. Another program, MAKE, automatically keeps programs up to date as components change. Two other programs, TXTCOM and SEDIT, function together to ease the task of maintaining parallel sources on different systems.

Control of changes provides key

Controlling access to a file, and recording any change to it, is the essence of a source-control system. With SourceCon, a file placed under source control becomes the base version. As the software is developed, each change to the base file is recorded separately, with a clear description of the person making the change, its purpose, the date and time, and the actual textual change.

Any version may then be reconstructed. Developers are able to generate releases from source-controlled files while simultaneously developing later versions of the same files. Developers are able to control the branching of software or documentation into several similar products without a proliferation of redundant files.

In addition, the clear audit trail helps developers isolate and correct human errors. In the case of large-scale mistakes, developers may replace the latest version of a file with an earlier one. Further, only one developer at a time has "write" access to source-controlled files, preventing conflicting changes from being made — a common cause of confusion and product delay in multi-programmer development.

'MAKE' rebuilds programs

A second component of SourceTools is MAKE, a file rebuilder that prevents the accidental inclusion of outdated modules into a large program compilation. After reading a user-written file describing the dependencies of each module on another, MAKE checks the date and time of each module and automatically rebuilds any that are out of date with respect to others. This facility improves program reliability and frees programmers for other tasks.

MAKE minimizes the number of commands required to rebuild programs and makes explicit the file dependencies.

Maintenance eased

The third major portion of SourceTools consists of two programs that, used together, automate the maintenance of parallel source files. TXTCOM compares the contents of two files and generates a script of the differences between them; SEDIT, a stream editor, reads the TXTCOM edit script and applies the necessary changes to one of the parallel source files. Because only the script of differences has to be transferred, these programs minimize the time required for updating sources at remote sites or on different processors.

SourceTools is available from Oregon Software or its authorized distributors. Documentation includes a 75-page user manual.

Pascal-2 moves to DEC's Pro 350

Beginning April 1, Oregon Software and authorized distributors are offering the Pascal–2 optimizing compiler for the DEC Professional 350, Digital's most advanced personal computer.

Developers may write Pascal–2 programs directly on the 350 under the RT-11 system. They also may use the 350 as a target machine, writing programs on the VAX or the PDP-11 and transfering the code to run on the 350.

The Pro 350 is a familiar environment for professional programmers. Many of them were trained on the 350's underlying PDP-11 hardware; Pascal–2 is the most widely used Pascal on the PDP-11. This means that a large body of applications programs may be moved to the 350 rapidly.

Standard language allows integration

The Pascal–2 language implementation is standard across all Digital operating systems, plus UNIX. Pascal–2 for the Pro 350 supports all capabilities of standard Pascal, including conformant array parameters, and code developed under Pascal–2 may be moved to other operating systems that have standard Pascal compilers.

Pascal–2 offers low-level integration with the RT-11 system. Pascal–2 programs may call subroutines written in Pascal or assembler, allowing the user to take advantage of existing system software. Pascal–2's language extensions provide sophisticated I/O handling and access to system-specific operations.

Tools included as options

Pascal–2 for the 350 includes, as an option, the full set of supporting tools available on other Pascal–2 systems: an interactive source-level debugger, an execution profiler, program and text formatters, and cross referencers for program analysis.

License fees for the complete Pascal–2/350 system are: $450 for the compiler and support library, including 90 days' warranty, documentation, sample program, and subscription to our newsletter. A package containing the debugger and utilities is $350. Software updates are $300 per update. All prices are U.S.

Documentation includes the standard *Pascal-2 User Manual.*

VAX cross-compiler targets any 68000 hardware, includes real-time capability

Oregon Software's 32-bit, VAX-based Pascal–2 system for developing applications software to run on the Motorola MC68000 is now available.

The cross-development software allows users to target applications, including real-time programs, for virtually any 68000 hardware configuration. Most of the development effort takes place in Pascal, with development on the 68000 limited largely to final testing.

The cross-development system consists of the optimizing Pascal–2 cross-compiler, which supports the international standard; a library of routines allowing user programs to run without an operating system on the 68000 and providing the capability for concurrent programming; a cross-linker/assembler producing loadable code for the 68000; and program development utilities such as PASMAT and XREF.

The VAX/VMS cross-compiler supports 32-bit integers. A 16-bit version was released earlier for RSX on the PDP-11.

The Pascal–2 cross-compiler performs extensive error checking and uses the same optimization techniques as the other Pascal–2 compilers to produce small, fast code. Pascal–2 supports conformant array parameters, adhering to Level 1 of the international Pascal standard.

Concurrency supported

Control of real-time processes is offered via the Concurrent Programming Package, which is an option to the system. The package enables users to develop all parts of an embedded or stand-alone system, even device drivers, in standard Pascal. The package offers true priority scheduling with a minimum of overhead.

The package consists of a library of procedures and functions that allow concurrent programming in the "process-monitor" style. This allows you to apply structured programming techniques to the problem of synchronizing the exchange of shared data among separate, concurrently executing processes.

The library also provides the interface with the hardware so that programs may run without an operating system. The library can be adapted to any hardware configuration; source code for all modules is provided to allow maximum flexibility when implementing the system.

Available now

The development system is now available from Oregon Software or its authorized distributors; the exact price depends upon the options selected.

32-bit VERSAdos native announced

In conjunction with release of the cross-compiler systems, Oregon Software also is announcing a native VERSAdos 32-bit compiler for the MC68000. This package includes an interactive source-level debugger, an execution profiler, and the other standrd Pascal–2 utilities such as formatters and cross-referencers. The compiler generates code compatible with the Motorola linker/assembler and also has the Concurrent Programming Package as an option.

How to read/write unformatted FORTRAN files from Pascal-2 under RSX-11M

by Joerg Donandt and Gerald Jahn

FORTRAN unformatted files are an efficient means of storing large data structures with respect to both storage and execution time. Unformatted data files require less storage space because redundancy is minimal. Conversions between internal binary form and a form readable by man (e.g., ASCII format) are unnecessary. For these reasons, we opted to use FORTRAN unformatted files to store digitized pictures, but we found that we needed to find a way to read and write them from Pascal programs — specifically those compiled with the Pascal–2 compiler. This article contains our solution and two samples of its use in Pascal programs.

Unformatted file structure

The way data are stored and accessed in unformatted FORTRAN files causes the difficulty. Data are stored in variable-length records (132 bytes maximum length) that may wrap around block boundaries. Normally, carriage-control characters determine the way that the FORTRAN files are printed out, but the file management system cannot distinguish between a normal FORTRAN print file, generated using a `WRITE(6,100)...` statement where 100 is the format statement for the write, and an unformatted file generated by `WRITE(3)...` statement.

Access to these records depends upon a two-byte length counter, which isn't normally available to the Pascal programmer. When accessing these records, the 132 bytes are read into the run-time system's buffer as a "segment," and every segment begins with a two-byte segment descriptor that indicates the segment's place in the file by one of four values:

Value	Description
1	first segment of the file,
2	last segment of the file,
3	only segment of the file,
0	any other segment between the first and last segments of a file.

Every filled record holds 130 data bytes, 2 segment-descriptor bytes, and 2 length-counter bytes, for a total of 134 bytes per record.

In the 130 bytes following the segment descriptor, data are interpreted according to the variable-type and the sequence in which those variables were written into the file. For example, a FORTRAN expression such as `INTR*2-VALUE` is stored as two data bytes, with the byte representing the least significant digits first.

Multidimensional arrays, such as those in the sample programs, are stored in a "first-index-quickest" fashion: the elements of an array are stored in adjacent locations, the first index is incremented for each element stored until its range is exhausted, then the second index is incremented. Thus, a matrix `MAT(I,J)` is stored with the elements indices, as shown:

```
(1,1)(2,1)...(I,1)
(1,2)(2,2)...(I,2)
  :     :       :
(1,J)(2,J)...(I,J)
```

The storage required to hold a data item such as our picture-matrix (named `Pic` in the two sample procedures) would have to be declared `DIMENSION BYTE MAT(128,128)` consisting of 128 * 128 bytes = 16384 bytes, stored in 130-byte segments. The file consists of 126 filled segments and one 4-byte "appendix," for a total of 127 segments.

For such calculations, you need to be aware of the following numbers: every filled record holds 130 data-bytes, 2 segment-descriptor bytes, and 2 length-counter bytes, for a total of 134 bytes per record. The last record consists of 4 data-bytes and a 4-byte overhead, for a total of 8 bytes. The file's length is calculated as:

$$134\ \textit{bytes/record} * 126\ \textit{records} + 8\ \text{bytes} = 16982$$

These are stored in 33 decimal blocks (44 octal), with each block containing 512 bytes, leaving 4 bytes empty in the last block.

Reading unformatted files

In order to read unformatted FORTRAN files, you must know **how** the unformatted file was written. With this in mind, look at sample procedure 1 (**Pic_read**). The file was generated by the FORTRAN statements:

```
DIMENSION BYTE MAT(128,128)
    :
WRITE(3) MAT
    :
```

From the above we know that:

- The file holds only one variable: **MAT**.
- The variable is structured as a two-dimensional array, with indices ranging from 1 to 128.
- The elements of this array are byte-values: they require only one byte storage per element.
- The elements are stored in standard form as previously explained.

To express these storage and structure characteristics in Pascal, we define the segment as:

```
type
  Byte = 0..255;
  Pic = packed array[1..128,1..128] of byte;
```

Note that we use a packed array. Since **Byte** is a subrange of **integer**, each element of **Pic** would normally be allocated two bytes of storage because **integers** are stored in two bytes. The packed array, however, forces the compiler to allocate only the storage needed, in this case, one byte.

The procedure does three basic operations: opens the appropriate unformatted file; reads a record from it; and transfers the data bytes into the **Pic** matrix. The procedure repeats the loop of reading and transferring until the matrix is filled.

The **reset** statement must contain the following file attributes:

/VAR:132 Variable-length record (maximum 132 bytes).

/NOBLK Records may wrap around block boundaries.

/FTN Use FORTRAN's carriage-control convention.

In **Pic_read** we have used the **read** statement for non-text files, violating the standard. The end-of-file (**eof**) does not work with non-text files, so we have introduced our own marker (**Endf**).

The statement **case** will handle the different possible segment-descriptors (**Code** in this procedure). The statement **code: for 3** would be somewhat inconsistent with our knowledge of **MAT**. Since **MAT** is large (about 16K Bytes), it must require more storage than a single record can provide. **Pic_read** therefore aborts for **Codes** not in the range 0 to 2. The **FOR K:=1 TO 130** loop empties the **Data** section of the segment, byte by byte, and stores the data at the appropriate position (indexed by **I,J**) in the **Matrix**.

The procedure is straightforward, and, once you've understood the details, you can modify it easily to fit your needs.

A tricky approach, however, must be used in writing unformatted FORTRAN files from Pascal.

Writing unformatted FORTRAN files

When we first tackled this problem, we had extreme difficulties writing segments into a variable-length record, non-text file. Indeed it was impossible. So we circumvented the problem with the following trick.

Instead of writing to a non-text file of variable-length records, we specify a normal text file and, using the **chr** function, we suppress the formatting normally done by the **write** statement. In addition, we used the **/FTN** attribute so that the FORTRAN run-time system (FOROTS) can find the attribute set when it tries to read unformatted data from the file.

Sample procedure 2 (**Pic_write**) does not write the file in portions of "records" but byte by byte. It starts by writing the segment-descriptor as two bytes (low byte first). Then it fetches a byte from the matrix (indexed by **I,J**) and writes it to the file. When it has written 130 bytes, a **writeln** lets the system store the record away and start a new one. During the **writeln**, the file buffer is emptied and the length-counter (mentioned in the preceeding discussion of file structure) is written to the file. Earlier, the length-counter could not be written, probably because the programmer is not allowed to use **writeln** statements on non-text files.

If you need to read/write unformatted FORTRAN files from the Pascal environment you ought to have a sound knowledge of the way both compilers treat your data. Although a general solution cannot be given, we hope that these remarks can help you solve your problem.

Mr. Donandt and Mr. Jahn are engaged in optical/digital image processing for pattern recognition purposes at The Institute of Applied Physics 1, University of Heidleberg, in Germany.

Sample Procedure 1

```
procedure Pic_read(fname: string;
                   var matrix: pic);

  type
    vlr = record   { a FORTRAN variable-length record }
      code: integer;
      data: packed array [1..130] of byte;
      end;

  var
    i, j, k: integer;
    datei: file of vlr;
    nxtrec: vlr;
    endf: boolean;

  begin
    endf := false;
    reset(datei, fname.ch, '/var:132/noblk/ftn');
    while not endf do
      begin
      read(datei, nxtrec);  { non-standard read }
      with nxtrec do
        begin
        case code of
          0: ;
          1:
            begin
            i := 1;
            j := 1;
            endf := false;
            end;
          2:
            endf := true;
          end;  { abort, if another CODE occured }
        for k := 1 to 130 do
          if j < 129 then
            begin
            matrix[i, j] := data[k];
            i := i + 1;
            if i > 128 then
              begin
              i := 1;
              j := j + 1;
              end;
            end;
        end { of with } ;
      end { of whilf } ;
    close(datei);
  end;
```

Sample Procedure 2

```
procedure Pic_write(Fname: string;
                    var Matrix: pic);

  var
    i, j, k: integer;
    datei: text;
    endf: boolean;
    c: byte;

  begin
    endf := false;
    i := 1;
    j := 1;
    rewrite(datei, fname.ch, '/var:132/noblk/si:-1/ftn');
    while not endf do
      begin
      c := 0;
      if (j > 127) and (i > 124) then
        begin
        endf := true;
        c := 2;
        end;
      if (j = 1) and (i = 1) then
        c := 1;
      write(datei, chr(c), chr(0)); {segment descriptor}
      for k := 1 to 130 do{ fill a record into MATRIX }
        begin
        if j < 129 then
          begin
          write(datei, chr(matrix[i, j]));
          i := i + 1;
          if i > 128 then
            begin
            i := 1;
            j := j + 1;
            end;
          end;
        end;
      writeln(datei);
      end;
    close(datei);
  end;
```

Letters

To the Editor:

The following is a plea to whomever (or whatever) has control over the release of existing errors in the Pascal–2 compiler. I realize that there may be a bit of pride which must be swallowed to admit that there is a bug in one's software but it must be done. Anyone familiar with the complexities of a compiler, much less one that performs such significant optimization, realizes that such bugs must exist but should be able to accept that.

As a user of Pascal–2, I would have to say that finding a compiler error is very frustrating. It is not until I have spent several days tracing down the bug, only to find that when I call one of your very helpful software support personnel that it is a bug which has been reported and known about, that I become very disappointed in the product. If I felt that I was the first person to encounter the error or had some chance to avoid it had I known about it, I would feel much more forgiving. (Obviously not admitting that it has been reported and resolved would be unforgivable.)

Of course we would all like every software product to be perfect but it isn't, not yet at least. So why kid ourselves? Let's communicate. The savings in real dollars as well as goodwill, I feel, would greatly outweigh the pride and postage it might cost you.

For instance, a compiler error could take as much as a day to track down to determine that it is in fact the

compiler and not a program error. If a programmer is paid nine dollars an hour and puts in an eight-hour day tracking the error down, it would be 72 dollars wasted. Multiply that by 100 installations which might come across the same error and already you have potentially wasted $7,200. Don't forget to take into consideration the friction created between the EDP managers, their programmers and Oregon Software.

All this could have been avoided with a simple monthly letter to all supported sites which would describe the problem and the way, if any, to code around it. I realize that compiling such a letter may be non-trivial but its benefits out here in user land would outweigh that significantly.

One other request I have is for a fast single-pass version of the compiler with, perhaps, optional code gereation. This again would create a great savings for us out here who must wait for 8 minutes, and sometimes more, for compiles on our meager 11/34s only to discover a missing variable declaration.

Both of these requests I make with all sincerity and hope they will be considered with the same degree of seriousness.

Sincerely, Harry A. Levinson

Editor's Reply:

Your plea for notice of known bugs has been heard, and we are in the process of improving our response. We are now better able to describe and track reported errors in a useful manner and hope to find a quick and painless way for users to keep up to date. We'll print a statement of the new policy in a future issue of the newsletter.

We have no plans for a single-pass version of the Pascal–2 compiler.

OPUS Communiqué

Oregon Pascal Users Society

Members should have begun receiving their membership list and member survey sheets on October 14, 1983 — just in time to contact others in the OPUS for Halloween.

The OPUS library will be opening soon. The logon for the OPUS library will be:

```
HELLO OPUS/LIBRT
    or:
HELLO OPUS/LIBRSX
    or:
HELLO OPUS/LIBRSTS
```

When you log on, a message appears:

```
For more information,
type SYSMSG.LST.
```

This file contains enough information to get you started on using the OPUS library. The OPUS library will be a read-only UFD. Dial-up lines are being installed now and the numbers should be in the October membership packet.

To put a program in the OPUS library, you must submit the source file only, with documentation of what it does, how to use it, how to compile and task-build/link it, and what its inputs and outputs are.

Everything submitted will be put into the library as long as it is consistent with OPUS goals and concepts. Send your routines to:

Bruce Williams,
OPUS Coordinator
c/o EOCOM
15771 Red Hill Ave.
Tustin, CA 92680
(714) 730-5051, ext. 302

Some have expressed interest in having a DECUS-like meeting. Such a meeting is really time-consuming and requires a lot of effort. All that can be said at this time is maybe and only if there is more interest.

In the meantime, we will "nybble" on the membership list problem. More information will be in the next Communiqué.

Accessing global symbols from Pascal programs

By Steve Poulsen

Many customers have asked how to access global symbols from a Pascal program. These are the global symbols used by the Linker and Task Builder, not the global-level variables defined at the start of a Pascal program. Users commonly want a data table defined in an assembler program from Pascal. It can be done, but it is tricky.

The concept of external data does not exist in Pascal, but the concept of external procedures does. The compiler generates a reference to the entry point of an external

procedure, using the first six characters in the procedure name as a global entry point. We use the compiler's ability to generate the address of a procedure to obtain the address of an external procedure.

When a procedure is passed as a parameter to another procedure, two parameters are passed. The first parameter is the static link used to access intermediate-level data in the proper context by the called procedure. Don't worry about it, we won't need to use it. The second parameter is the actual address of the procedure in memory. This is what we are after. Now all we need to do is to obtain the value of such a procedural parameter.

You must use an external procedure to obtain these two parts of a procedural parameter without violating Pascal's type checking rules because parameter typing cannot be performed across module boundaries. Define a module called GETADR.PAS (see example), a very simple function that takes two integer parameters and returns the second parameter as the function result. The trick is that we are really going to pass it a procedural parameter, and it will return the address of the entry point of the procedure. This function must be compiled as an external procedure to avoid the compiler's type checking rules.

The secret of accessing global symbols is to define the symbol as an external procedure and pass that procedure to a function that returns the address of the variable. For example, suppose that you want to access the variable **$DSW**, which is the RSX directive status word whose location is defined by the Task Builder. You include the function **Getadr** in a program as shown in the figure below. When the program is compiled and linked with the GETADR module (shown above the program), it prints the current value contained in the global variable **$DSW** (usually a 1).

```
{$nomain}
program Getadr;
{  Return the address of a procedure}

  function Getadr(static_link, entry_point: integer ): integer;
  external;

  function Getadr;
    begin
      Getadr := entry_point;
    end;


program Print_Dsw;

type
  Pointer = ^integer;

var
  Dsw_pointer: pointer;  { Address of DSW }

procedure $Dsw;
  external;

function Getadr(procedure Magic): integer;
  external;

begin
  Dsw_pointer := Loophole(pointer, getadr($dsw));
  writeln(''$Dsw = '', DSW_pointer^);
end.
```

Meeting design goals for Pascal-2 Concurrent Package

by Michael S. Ball

As computers increase in power and drop in price, many individual processors are being included as components of larger systems. These component processors are called embedded systems, and they are typically used for process control, data logging, communications processing, and signal processing. Such computers must deal with unique I/O devices and frequently must respond to external events in a short time. Traditionally, such applications have been coded in low-level languages, usually the assembly language for the machine. A part of the same tradition recognizes that such applications are difficult to code and even more difficult to debug. An efficient approach to concurrency is crucial to a successful embedded system.

Concurrency models

Two basic models are used in writing concurrent programs. A "message model" looks at a concurrent program as a collection of processes connected by message queues. The processes communicate by passing messages between themselves. A "procedure model" sees a concurrent program as a group of processes that communicate through procedures and shared memory. The two views are in fact isomorphic, and any concurrent program written using one viewpoint can be rewritten using the other.

The difference appears to the programmer in the "primitives" supported by the systems. The primitives of a message-passing system are usually variations on two types: "send a message" and "receive a message." Special-purpose primitives may also exist, such as "send a message and wait for a reply," but the only required primitives are "send" and "receive." The primitives in a procedure-based system

are "acquire access to shared memory" and "release access to shared memory" (historically called "P" and "V"). Again, specialized versions may be implemented, but the only neccessary ones are "acquire" and "release."

Each set of primitives may be implemented in terms of the other set and neither has any "special ability" not shared by the other. So how do we pick the approach to use? We look at the environment and applications intended for the system.

Let's consider two hypothetical examples. The first system consists of a number of microprocessors, each containing its own memory. Each microprocessor executes a single process, and the processes communicate over I/O channels. In this system, a message-passing scheme maps directly onto the hardware and is the obvious choice. The procedure-based system would have to simulate shared memory with messages, a profitless venture. In the second system, one or more processors share a common memory. Each processor may execute one or more processes, but large amounts of data are passed between processes. A procedure-based approach would be more appropriate in this system because communication takes place using shared memory anyway and a message-based system requires copying the data as it passes between processes.

An additional factor to consider is the type of I/O devices to be handled. The usual minicomputer or microcomputer device is easily treated as a process that shares memory with internal processes and communicates using signals. Devices with separate channel controllers may well fit more easily into a message model.

Our design goals matched better with the procedure model, so the Concurrent Programming Package supports that model of concurrent programming. The process-monitor approach of the Concurrent Package is based on a model that is slightly more refined than the basic one described above.

Process-monitor model

Process-monitor style concurrent programming was initially proposed by C. A. R. Hoare [1979] and has been used in the programming languages Concurrent Pascal [Hansen 1977] and Modula [Wirth 1977].

The objective is to isolate all communication between processes into *monitors*. Each monitor consists of some shared memory (monitor variables) for communication and a set of procedures to manipulate this shared memory. Each queued procedure executes an "acquire access" operation on entry and a "release access" operation just before exit. All manipulations of the shared memory must take place within these monitor procedures, guaranteeing consistency of the data.

If a procedure within a monitor has to wait for some external condition, it does a "release access" operation for the monitor variables and waits — using a variation of "acquire access" — for some other process to make the condition true.

Although monitors need only "acquire" and "release," specialized forms of these operations improve efficiency and programming ease.

The Pascal–2 Concurrent Programming Package uses two special operations, **lock** and **unlock**, to control access to monitor variables. Each monitor has a special variable, a *gate*, that is used as an argument to **lock** and **unlock**. The monitor code calls **lock** on entry to the procedure and **unlock** on exit. Without a separate gate for each monitor, entry into any monitor would lock out entry into all other monitors.

The special operations **wait** and **signal** provide synchronization between processes. The process that executes **wait** unlocks the monitor variables and suspends execution until a **signal** is executed by some other process. **Wait** and **signal** operations are controlled by special variables called *events*. One speaks of "waiting for an event" or "signaling the occurrence of an event."

Device interface

Embedded systems frequently have to drive a variety of I/O devices, and a useful concurrent programming system must provide support for device drivers. A device handler requires three things of the system:

- Access to the control registers for the device. On DEC's PDP-11 and Motorola's MC68000, device registers are mapped into the address space and manipulated like any other location in memory. The **origin** feature of the Pascal–2 compiler may be used to place a variable at the physical address of a device register; the register is then manipulated with normal assignment statements. Some additional procedures may be necessary for a computer that uses special I/O commands.

- Synchronization with the external device. Since external devices provide truly concurrent execution, even in a single-processor system, the hardware usually provides interrupts to signal that execution is completed. We can make the interrupt perform a "signal" operation as though it were an internal process.

- A way of providing mutual exclusion for access to the shared variables. The usual way of handling this in a computer is to turn interrupts off when manipulating data that is shared with the external device.

The Concurrent Programming Package meets the last two requirements with two new primitives. The first one is **makedevice**, which takes a gate as an argument and specifies

that the calling process is to execute with interrupts off. In addition, any process executing a **lock** on that gate has interrupts off until it executes a **wait** or an **unlock**. The process that calls **makedevice** is called a "device process."

Any device process can execute the **intwait** primitive, which has the same effect as a **wait** except that the signal is provided by an external device. During the time a process is waiting for an interrupt, other processes may execute within the monitor and have access to the shared memory.

The device process is sometimes called a "shadow process" for the external device. The combination can be considered a single process that executes on the external device during the **intwait** period and on the central processor at all other times.

Implementation decisions

The simplest way to achieve concurrency on a single processor is the use of coroutines. The processes in a coroutine system execute one at a time, changing control only at specific points in the code. Using of the primitives described above, the control changes from one process to another only at calls to **wait** and **signal**. **Lock** and **unlock** primitives become unnecessary, as control changes only when the monitor gate would be unlocked anyway.

Device processes can be incorporated in a coroutine system quite simply. The **intwait** primitive saves and restores registers so that the execution of the device process has no direct effect on other processes. When the device process executes a signal that actuates an ordinary process, control does not pass directly to that process. Instead, the process is made ready and control passes to it at the next **wait** operation performed by an ordinary process.

The coroutine system is attractive in its simplicity. The overhead is minimal; a process swap costs little more than an ordinary procedure call. If the primitives are built into the compiler, the routine approach doesn't even need to save registers across a swap. On the other hand, response to I/O is delayed until a process voluntarily relinquishes control. A task with long periods of computation between interactions can slow response drastically.

A full concurrent programming scheme should also provide for process priorities and preemption of running processes. In such a system, process swapping occurs whenever a process with a higher priority than the current process becomes ready to execute. The disadvantage of such a system lies in the increased frequency of process swapping and the added complexity of each swap. Careful implementation can minimize the frequency of process swapping, however, and the added complexity is a small price to pay if you need to combine long computations with fast response.

Our implementation

The Concurrent Programming Package provides Pascal–2 programmers with a set of primitives for programming embedded applications. The package supports multiple processes, device interfaces, and synchronization of processes. The entire application, from device drivers up, can be coded in standard Pascal.

The design goals for the package were to provide:

- Support for a variety of embedded systems. This includes a potentially large number of concurrent processes, some of which may have long sections of computation with interactions between other processes.

- Resultant software or programs able to run on a single mini or micro. Multiple processor systems are not difficult, but the coding details are dependent on the exact architecture of the system.

- Ability to write system-specific device handlers in Pascal.

- Minimal response time to external events and reduced cost of interprocess communication, so that it is cheap enough to be used whenever it fits the application.

- Resulting code capable of operating in a mixed ROM/RAM environment.

- A small, modular package. If an application does not require a particular capability, it should not have to pay the price for it.

- A self-instrumenting package that gives the programmer complete information on the state of the system when an error occurs.

- The ability to gather sufficient data on a program to spot and eliminate bottlenecks.

The Pascal–2 Concurrent Programming Package provides a scheme for multiple process priorities together with preemption. The extra cost is small compared to the capability gained. Process swapping is extremely efficient, with about 20 instructions executed in the worst case. This is little more than a simple procedure call.

The package includes a comprehensive set of diagnostic tools, including a process-by-process walkback in case of error. A log of the most recent primitive calls aids in the detection of subtle synchronization problems. The statistics gathered for each gate show the number of times the gate is used and the number of conflicts arising from two processes attempting to use the same monitor at the same time. Such statistics allow the detection of major performance bottlenecks.

The package meets all of the design goals and provides a useful addition to the software toolbox of system developers.

References

C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, Vol 17, No. 10, October 1974.

Per Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1977.

N. Wirth, "Modula, A Language for Modular Multiprogramming," *Software Practice and Experience*, Vol 7, No. 1, 3-35 (1977).

Mr. Ball designed and implemented the Concurrent Programming Package at Oregon Software. Upon leaving Portland for a sunnier clime, he founded TauMetric Corporation, where he is a consultant for custom system software development. Address: 1094 Cudahy Place, Suite 302, San Diego, CA 92110, (619) 275-6381.

Implementing real-time facilities in Pascal

by Hilding Elmquivst and Sven Erik Mattsson

Introduction

The increasing use of microcomputers in technical systems has made the art of real-time programming more important. We have previously used Concurrent Pascal [Brinch Hansen, 1971] in courses on real-time programming. However, we needed more flexibility. For example, we wanted to be able to demonstrate message-passing schemes and rendezvous. Furthermore, in order to be able to give the students a profound understanding of how concurrency is achieved, the nucleus or kernel of such a system has to be shown.

A natural way of introducing concurrency is to start with a sequential language such as Pascal and add necessary routines without changing the compiler or the support library. Per Brinch Hansen [1978] and Kriz and Sandmayr [1980] give descriptions of such routines in Pascal-like notations. However, it is not always possible to write these routines in Pascal because hardware facilities such as registers must be manipulated when the processor is switching between processes. Brinch Hansen [1978] translated the kernel by hand to assembly language.

The real-time kernel we've developed supports concurrent programming in Pascal, particularly in Pascal–1. The kernel is written in Pascal, but it relies on a small number of assembly-language procedures for process creation, transfer between processes, and handling of interrupts. This set of routines is called the nucleus. The kernel implements semaphores for mutual exclusion and events for other synchronization. The kernel also provides the capability to program I/O handlers. With this real-time kernel, Pascal has the same expressive power as Concurrent Pascal. However, since the user's concurrent program is compiled with the standard Pascal compiler, there is much less security than in Concurrent Pascal.

In this article, we've described the implementation for Pascal–1 on the PDP-11 computer. The kernel has also been implemented on the Texas Instruments TMS 9900 and Motorola MC68000 processors.

Kernel primitives

Three primitives handle creation, scheduling, and execution of processes. Others perform duties such as communication between processes and synchronization of their execution, timer control, and interrupt handling.

Process handling

A process is declared as a parameterless procedure, which we refer to as "process-procedure" hereafter.

A process instance may be created by a call to **CreateProcess** from anyplace where the procedure could be called in the ordinary way:

```
procedure CreateProcess
  (procedure proced;
   memreq: unsignedinteger);
```

where *proced* is the process-procedure describing the process, and *memreq* is the memory requirement (in bytes) for the stack and the heap of the process.

The processes are scheduled according to their priorities. The priority of a process can be changed dynamically by calling:

```
procedure SetPriority
  (priority: integer);
```

where *priority* is the new priority of the process.

The main program must be converted to a process by a call to **InitKernel** before other processes can be created:

```
procedure InitKernel
  (memreq: unsignedinteger);
```

where *memreq* is the memory requirement (in bytes) for stack and heap (global variables excluded). The procedure **InitKernel** also creates a clock process and an idle process.

Communication

Communication of data between processes is done with variables that are accessible to the process-procedures according to scope rules. The programmer must ensure mutual exclusion by the use of semaphores, as follows:

```
procedure InitSemaphore
  (var sem: semaphore;
   initval: integer);

procedure Wait
  (sem: semaphore);

procedure Signal
  (sem: semaphore);
```

where *sem* is the semaphore and *initval* is the initial value for the semaphore.

The synchronization between processes, such as waiting for a condition on a shared variable, is done by using the concept of an event, introduced by Brinch Hansen [1973]. There are three operations on events:

```
procedure InitEvent
  (var e: event;
   sem: semaphore);

procedure Await
  (e: event);

procedure Cause
  (e: event);
```

where *e* is the event variable, and *sem* is the associated semaphore for mutual exclusion.

An event must be initialized by a call to **InitEvent**, which associates the event with the semaphore for mutual exclusion. A call of **Await** delays the process and makes an implicit signal to the associated semaphore. A call to **Cause** by another process moves all delayed processes to the queue of the associated semaphore.

Clock handling

The procedure **WaitTime** makes the calling process wait a specified time interval, which is expressed in ticks. A tick is 20 milliseconds.

```
procedure WaitTime
  (time: integer);
```

Interrupt handling

The procedure **WaitIO** makes the calling process wait for a specified interrupt. The procedure sets the enable bit in the specified status register before suspending the running process. Some other process is then scheduled for execution. When the interrupt occurs, the enable bit is reset.

```
procedure WaitIO
  (vecaddr: integer; var statusreg: integer);
```

The process waiting for the interrupt is indicated as ready for execution and the process having the highest priority is resumed. Only one process at a time can wait for a certain interrupt and this must be guaranteed by the user.

The PDP-11 has memory mapped I/O. Pascal–1 allows manipulation of the device buffers and status registers as ordinary variables by allowing specification of addresses in the variable declaration.

Program organization

Since the programmer has to ensure mutual exclusion himself, it is important to organize the program in a way that aids in using the semaphores correctly. A natural solution is to collect the data, the semaphore for mutual exclusion, and the event variables in a record. Operations on the data are then conveniently done via a **with** statement. Ordinary Pascal allows recursion, thus procedures are reentrant, so it is possible to construct a set of procedures that operate on the shared data. This is the idea behind the monitor concept in Concurrent Pascal. Pascal–1 allows **type**, **variable**, and **procedure** declarations to be mixed, making it possible to collect the record declaration and the procedures together.

Implementation

The introduction of concurrent processes means that code, processor and storage are shared resources. The problem of handling and protecting these resources must also be considered. Our aim is to use standard Pascal as far as possible. However, as hardware facilities such as registers must be manipulated, it is not possible to make the code completely portable. These computer-dependent parts are isolated in a small set of assembly procedures called the nucleus.

Hardware

The PDP-11 computer has eight 16-bit general registers R0 through R7 [Digital Equipment Corporation, 1976]. The register R6 normally serves as the stack pointer (SP), pointing to the top element. The stack grows downward in the memory. Register R7 is the program counter (PC) of the processor.

Pascal–1 run-time organization

Figure 1 shows the memory usage for a Pascal–1 program. All global variables are indexed relative to register R5. The

top-of-heap pointer is a global assembly variable called $KORE. The stack and the heap grow toward each other. When a procedure is called or an allocation is made on the heap, a run-time check for overflow is performed.

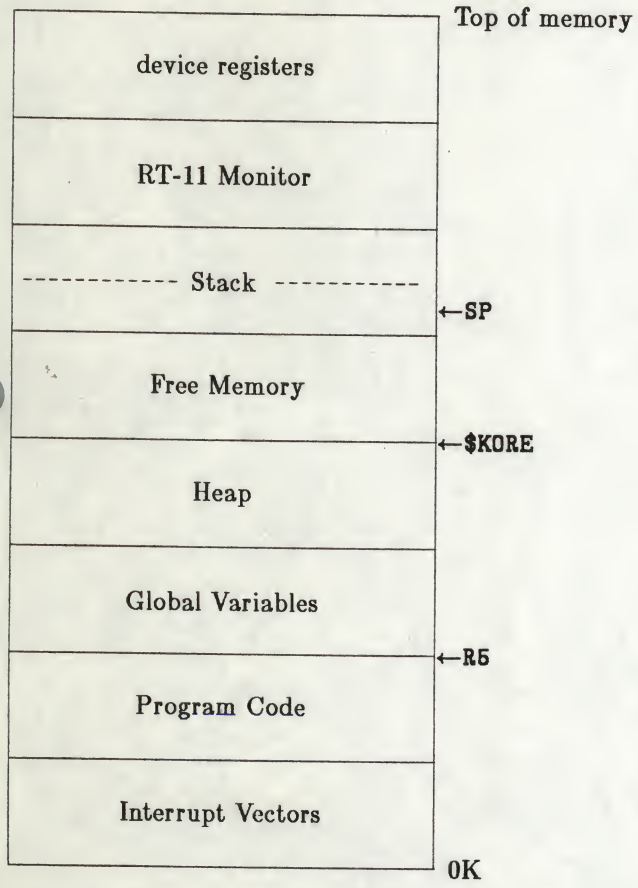

Figure 1. Pascal–1 program memory organization

When a procedure is called in a Pascal–1 program, the parameters are calculated and pushed onto the stack. The first parameter is stacked first. When all parameters are stacked, a jump to the procedure is done with the assembler instruction JSR PC, SUBR, which pushes the return address (PC) onto the stack. The procedure then allocates space on the stack for the local variables.

All local variables and the parameters are accessed by indexing with SP. Intermediate-level variables are accessed by the use of the static links. The static link for an invocation of a procedure is a pointer to the stack frame of the latest invocation of the lexically enclosing procedure. To access an intermediate variable, it is necessary to follow the static links until the proper stack frame is reached and index by that base value. Information on the new static link is kept in R4 during the subroutine jump. The static link is pushed onto the stack on top of the local variables (see Figure 2). The static link is not used in global procedures.

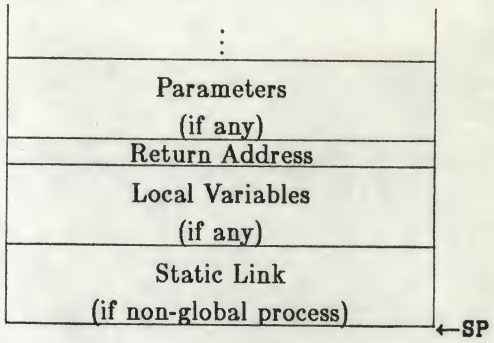

Figure 2. The stack frame

Memory and process handling

A basic requirement for concurrent processes is that they have separate stacks for local variables and stack frames of called procedures.

The main program is converted to a process by a call of the nucleus procedure **InitNucleus**.

```
procedure InitNucleus
  (memreq: unsignedinteger;
   VAR startfree, endfree: unsignedinteger;
   VAR main: process);
```

where *memreq* is the memory requirement (in bytes) for main process, *startfree* is the returned start address for free memory, *endfree* is the returned end address for free memory, and *main* is the process variable for the main process.

An estimate of the memory requirements for stack and heap of the main process must be given as *memreq*. Global variables are excluded but space for stack frames of procedures called by the main process should be included. The procedure also returns information about free memory. The information is used later when subprocesses are allocated memory.

A new process is created by calling **NewProcess**, the nucleus procedure, as follows:

```
procedure NewProcess
  (procedure proced;
   startarea, endarea: unsignedinteger;
   var proc: process);
```

where *proced* is the process-procedure for the process, *startarea* is the start address for stack and heap area, *endarea* is the end address for stack and heap area, and *proc* is the process variable for the process.

Figure 3 shows the assembly code for **NewProcess**.

```
;  procedure newprocess(procedure proced;
;    startarea, endarea:
;    unsignedinteger;
;    var proc: process);
;  Creates a process from the procedure proced
;  and associates it with 'proc'.

NEWPROCESS:
  MFPS  -(SP)           ; push(PSW)
  MTPS  #340            ; disable

  MOV   6(SP), R0       ; R0:=endarea {child.SP}
  MOV   8.(SP), R1      ; R1:=startarea
  MOV   R1, @4(SP)      ; proc:=R1
  MOV   12. (SP), R4    ; R4:=proced.staticlink

  MOV   10.(SP), -(R0)  ; pushchild(proced.startaddr);
  MOV   R4, -(R0)       ; pushchild(R4)
  MOV   R5, -(R0)       ; pushchild(R5)
  MOV   R1, R2          ; R2:=R1
  ADD   #2, R2          ; R2:=R2+2
  MOV   R2, -(R0)       ; pushchild(R2) {$KORE}
  MOV   #RESCHILD,-(R0) ; pushchild(RESCHILD)
  MOV   R0, (R1)        ; proc^:=R0 {child.SP}

  MTPS  (SP)+           ; pop(PSW)
  MOV   (SP), 10.(SP)
  ADD   #10., SP
  RTS   PC

RESCHILD:               ; {Start of child}
  MOV   (SP)+, $KORE    ; pop($KORE)
  MOV   (SP)+, R5       ; pop(R5)
  MOV   (SP)+, R5       ; pop(R4)
  MTPS  #0              ; enable
  JSR   PC, @(SP)+      ; proced
                        ; {shouldn't come here}
LOOP:
  JMP   LOOP            ; while true do;
```

Figure 3. Procedure NewProcess

Processes communicate by using variables that are accessible for the process-procedures according to the ordinary Pascal scope rules. There is no problem using global variables since they are all accessed relative to register **R5**. The parameter *proced* contains information about the static link and the address of the code for the procedure.

A process is suspended by a call to the nucleus procedure **Resume** or **IOresume** or by an interrupt. The context (relevant registers, **$KORE**) of the process is stored on the stack (see below) of the process. The stack pointer is saved just below the heap of the process. The address to this location is the value of the process variable. The memory area of a suspended process is shown in Figure 4. The nucleus has a copy of the process variable of the currently running process in an assembly variable named **CURRENT**.

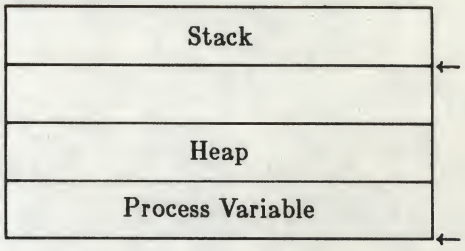

Figure 4. Memory area of a suspended process

The execution of a process **p** is resumed by a call to the nucleus procedure **Resume**:

```
procedure Resume
  (proc: process);
```

where *proc* is process variable of the process to be resumed. Figure 5 shows the assembly code activated by the call.

```
;  procedure resume(proc: process);
;  Resumes the process 'proc'.

RESUME:
  MFPS  -(SP)           ; push(PSW)
  MTPS  #340            ; disable
                        ; {Store context}
  MOV   R5, -(SP )      ; push(R5)
  MOV   $KORE, -(SP)    ; push($KORE)
  MOV   #RES, -(SP)     ; push(RES)
  MOV   SP, @CURRENT    ; CURRENT^ := SP
                        ; {Resume 'proc'}
  MOV   10.(SP), CURRENT ; CURRENT:=proc
  MOV   @CURRENT, SP    ; CURRENT:=proc
  JMP   @(SP)+          ;  switch

RES:                    ; {Restore context}
  MOV   (SP)+, $KORE    ; pop($KORE)
  MOV   (SP)+, R5       ; pop(R5)
  MTPS  (SP)+           ; pop(PSW)

  MOV   (SP), 2(SP)
  ADD   #2, SP
  RTS   PC
```

Figure 5. Procedure Resume

The nucleus procedure **IOresume** makes the current process wait for a specified interrupt and resumes another process:

```
procedure IOresume
  (proc: process;
   vecaddr: integer);
```

where *proc* is the process variable of the process to be resumed, and *vecaddr* is the vector address for awaited interrupt. When the interrupt occurs, the current process, which might not be *proc*, is suspended and the process waiting for the interrupt is resumed.

The LSI-11 has only two priority levels: interrupts enabled or disabled. The nucleus contains two procedures that change the status of the processor.

```
procedure enableinterrupts;

procedure disableinterrupts;
```

All processes, including the main program, start with the interrupts enabled.

Upon interrupt, the content of PC is automatically pushed onto the stack. PC is loaded from a preassigned memory location called an *interrupt vector*. The actual location is chosen by the device interface designer and is located in low memory address. When the specified interrupt occurs, the suspended process should be resumed. The desired effect of the interrupt is thus that a call would be done to a procedure:

```
IntResume(iosuspended: process);
```

Procedure **IntResume** is similar to **Resume**. The argument **iosuspended** is the process that called **IOresume** with the vector address corresponding to the interrupt. However, the hardware handling of the interrupts on a PDP-11 allows only the calling of a parameterless procedure when the interrupt occurs. This hardware facility is not convenient in connection with reentrant routines that are called by several processes.

One solution to this problem is to dynamically create, for each process, a small procedure that calls **IntResume** with the appropriate argument. In fact, the only thing that needs to be done is to place the assember instruction **JSR PC,INTRESUME** above the top of the stack of the process calling **IOresume** and the address to this instruction in location **vecaddr**. When the interrupt occurs and the **JSR** instruction is executed, the return address is stacked on the stack of the currently executing processes. However, the return address is exactly the stack pointer (i.e. the value of the process variable) for the I/O-suspended process. When the interrupt occurs, the process that should be resumed is made known to the interrupt handling routine in an efficient manner.

Processor management

The kernel primitives contain switches of the processor between the processes. The nucleus has procedures that can suspend and resume processes, but the kernel must decide which processes should be running. The kernel stores relevant information about the processes in records of type **processrec** (see Figure 6). The kernel keeps these records in different double-linked lists. Every list has a head of the same type as the rest of the elements in the list in order to avoid special handling of empty lists.

The processes that are competing for the processor are kept in **readyqueue**, and the variable **running** keeps track of the running process.

One list of process records is associated with each semaphore **waiting** and each event **delayed**. All processes waiting a specified time are kept in a single list **timequeue**. They are ordered according to the increasing waiting times. The process record contains one field **time**, which contains the waiting time relative to the preceding process in the time queue. The waiting time for the first process is relative to the current time. This is an efficient way of organizing the time queue because the clock process needs only to decrement the time field of the first process at each clock tick. The relative waiting times are calculated by the procedure **WaitTime**.

Only one process is allowed to wait on a specific interrupt. Therefore, no list of process records is needed. A reference to the process record of the waiting process is kept in a local variable in each instance of the procedure **WaitIO**.

The scheduling rules make it natural to order the elements in the ready queue and in the queues associated with semaphores according to their priorities. The order is unimportant in a queue associated with an event, so its elements can be inserted at the end.

Kernel structure

The data structure of the kernel is a shared resource and mutual exclusion is accomplished at this level by disabling interrupts.

The entry routines of the kernel must be visible from the user's program. The Pascal–1 compiler allows separately compiled modules with procedures and functions. A global routine in a separately compiled module is visible from other modules, if the compiler's external module switch is turned on ({**$E+**}). The user who wants to use a separately compiled routine must define a procedure heading followed by the keyword **external**. A file that contains predefined types and procedures declared external is available as a prefix to the user programs.

Conclusion

These techniques allow programmers to introduce concurrency in ordinary Pascal without changing the compiler or the support library, an approach suitable for educational purposes. The kernel is used in an undergraduate course in real-time programming. Programs for control of a physical process, operator communication and point-to-point protocol for computer communication have been implemented by the students. It is also possible to use the nucleus to test new primitives for concurrent programming. Routines for message passing and for the rendezvous concept of Ada have been implemented.

The resulting kernel is comparable to Concurrent Pascal or to Texas Instruments μP (Microprocessor Pascal). Hoppe [1980] presents a similar kernel with message-passing mechanisms written in Modula-2 [Wirth, 1980]. Some extensions have been made to the kernel in order to make it easier to work with. We've used a D/A converter, which outputs an analog signal that is displayed on an oscilloscope, to show what process is currently running. It is also possible to take a snapshot and generate a status report for the processes. A multi-terminal handler has been written in Pascal. It has been connected to the **read/write** statements of Pascal by handling the software interrupts generated by the support library.

The nucleus (Figure 6) shows what has to be added in order to obtain concurrency in Pascal. The code is written in a straightforward way to make it easy to understand. The primitives introduced in the nucleus are similar to those of Modula-2. Interrupt handling is included in an efficient way.

References

P. Brinch Hansen, *Operating System Principles* (Englewood Cliffs, New Jersey: Prentice Hall Inc., 1973).

P. Brinch Hansen, *The Architecture of Concurrent Programs* (Englewood Cliffs, New Jersey: Prentice Hall Inc., 1978).

Microcomputer Handbook (USA: Digital Equipment Corporation, 1976).

H. Elmqvist and S. E. Mattsson, "Implementation of Basic Primitives for Concurrent Programming in Pascal," (Lund, Sweden: Department of Automatic Control, Lund Institute of Technology, CODEN:LUTFD2/(TFRT-7230)/1-023/, 1981[a]).

H. Elmqvist and S. E. Mattsson, *A Real-Time Kernel for Pascal* (Lund, Sweden: Department of Automatic Control, Lund Institute of Technology, CODEN:LUTFD2/(TFRT-7[illegible])/1-023/, 1981[b]).

J. Hoppe, "A Simple Nucleus Written in Modula-2: A Case Study," *Software Practice and Experience* 10 (1980): 697-706.

J. Kriz and H. Sandmayr "Extension of Pascal By Coroutines and Its Application To Quasi-Parallel Programming And Simulation," *Software Practice and Experience* 10 (1980): 773-789.

Pascal-1, Version 1.1 for RT-11 (Portland, Oregon: Oregon Software Inc., 1978).

N. Wirth, *Modula-2* (Zurich, Switzerland: Institut fur Informatik, ETH, 1980).

Professors Elmqvist and Mattsson teach real-time programming in the Department of Automatic Control at the Lund Institute of Technology in Sweden. They have designed a kernel that supports concurrent programming in Pascal–1. Their approach shows what must be added to set concurrency in a sequential language and it is therefore suitable for education. This description of their real-time kernel was presented at the IFAC Symposium on Software for Computer Control 1982 and is reprinted here by permission of the International Federation of Automatic Control. The authors want to thank Leif Andersson for many good ideas and stimulating discussions.

```
{Real-Time Kernel for Pascal.}
{Use the nucleus prefix.}

CONST maxpriority = 1000;

TYPE unsignedinteger = 0..65535;
     processref =^processrec;
     semaphore =^semaphorerec;
     event =^eventrec;
     processrec = RECORD
     succ, pred: processref;
     proc: process;
     priority: integer;
     time: integer;
     END;
     semaphorerec = RECORD
     counter: integer;
     waiting: processref;
     END;
     eventrec = RECORD
     reentry: semaphore;
     delayed: processref;
     END;
VAR  running, readyqueue, timequeue: processref;
     freetop, freebase: unsignedinteger;

PROCEDURE put(p, q: processref);
{Inserts process record p before process record q in q's list.}
  BEGIN
  p^.succ:=q;
  p^.pred:=q^.pred;
  q^.pred^.succ:=p;
  q^.pred:=p
  END;

PROCEDURE remove(p: processref);
{Removes processrecord p from its list.}
  BEGIN
  WITH p^DO
    BEGIN
    pred^.succ := succ;
    succ^.pred := pred
    END
  END;

PROCEDURE putpriority(p, q: processref);
{Inserts processrecord p in queue q according to priority.}
  VAR p1: processref; pri: integer;
  BEGIN
  pri := p^.priority;
  p1 := q^.succ;
  WHILE (p1 <> q) AND (pri )= p1^.priority)
    DO p1:=p1^.succ;
  PUT (p,p1)
  END;
```

```
PROCEDURE schedule;
  BEGIN
  IF readyqueue^.succ <> running THEN
    BEGIN
    running := readyqueue^.succ;
    resume(running^.proc);
    END;
  END;

{$E+} {external}
PROCEDURE createprocess(PROCEDURE proced; memreq: unsignedinteger);
  VAR child: processref;
  BEGIN
  disableinterrupts;
  freetop := freetop - memreq;
  new(child);
  child^.priority := 1;{Default priority.}
  putpriority(child, readyqueue);
  newprocess(proced, freetop, freetop+memreq, child^.proc);
  schedule;
  enableinterrupts;
  END;

PROCEDURE initkernel(memreq: unsignedinteger);
  CONST clockarea = 100; idlearea = 100;
  BEGIN
  disableinterrupts;
  {Create readyqueue with running.}
  new(running);
  new(readyqueue);
  readyqueue^.succ := running;
  readyqueue^.pred := running;
  running^.succ := readyqueue;
  running^.pred := readyqueue;

  {Create empty time queue.}
  new(timequeue);
  timequeue^.succ := timequeue;
  timequeue^.pred := timequeue;

  running^.priority := 1;
  initnucleus(memreq, freebase, freetop, running^.proc);

  createprocess(clock, clockarea);
  createprocess(idleproc, idlearea);
  END;
```

```
PROCEDURE initsem(var sem: semaphore; initval: integer);
  BEGIN
   new (sem);
   WITH sem^ DO
     BEGIN
     counter:=initval;
     new(waiting); {Empty waiting queue.}
     waiting.succ:=waiting;
     waiting.pred:=waiting;
     END;
  END;

PROCEDURE wait(sem: semaphore);
  BEGIN
  disableinterrupts;
  WITH sem^ DO
    BEGIN
    IF counter >0 THEN
      counter := counter -1
    ELSE
      BEGIN
      remove(running);
      putpriority(running, waiting);
      schedule
      END;
    END;
  enableinterrupts;
  END;

PROCEDURE signal(sem: semaphore);
  VAR p:processref;
  BEGIN
  disableinterrupts;
  WITH sem DO
    BEGIN
    IF waiting <> waiting^.succ THEN
      BEGIN
      p := waiting^.succ;
      remove(p);
      putpriority(p, readyqueue);
      schedule
      END
    ELSE
      counter:=counter+1
    END;
  enableinterrupts;
  END;
```

Figure 6. This listing contains some parts of the kernel. The procedures **SetPriority**, **Initevent**, **Await**, **Cause**, **WaitIO** and **WaitTime** and the processes **Idleproc** and **Clock** are not included.

The Log: Pascal-1 and Pascal-2

Oregon Software's previous Pascal Newsletter described the significant changes in Pascal–2 V2.1A. This log details bugs fixed in Pascal–2 V2.1B, which may now be ordered. Several fixes listed for V2.1 involve the support library; such fixes also apply to Pascal–1.

The changes described apply to all versions of Pascal–1 or Pascal–2 unless a specific operating system is specified. When possible, work-arounds are given for V2.1A users.

Pascal–2

Version 2.1B corrects these problems for all PDP-11 systems.

Debugger altered registers

The Debugger altered some of the program's registers when the `C`, `S`, or `P` commands were used to start a program running under the Debugger. Work-around: use `G` to begin program execution. After the program has started executing, the `C`, `S`, and `P` commands work correctly.

Negative zero

The `write` procedure sometimes printed a negative sign for negative real numbers that round to zero. For example, `write(-0.010:7:1)` printed the value −0.0. The correct result is 0.0. This error did not occur with all such numbers.

'Sin' and 'Cos' problem

The double-precision `sin` and `cos` routines did not return the correct value when their argument was exactly 0.0. Work-around: use a special check for the argument 0.0, as in the following example:

```
function Xcos(X:real):real;
begin
  if X = 0.0 then Xcos := 1
  else Xcos := Cos(x)
end;
```

Line numbers thrown off

The use of included files threw off error walkback line numbers. Work-around: use the new syntax, with quotes around the file name.

```
%include 'filename';
```

The old syntax may not continue to be valid in the future, anyway.

Debugger changed variables

When a fatal error is detected in a user's program, the Debugger regains control after the error message is printed. However, the Debugger did not print the correct values for some variables, because the error-reporting procedures in the support library did not preserve registers and the Debugger cannot print the values of variables stored in those registers. V2.1B fixes the problem for many run-time errors, but certain kinds of run-time errors may still change the values of some variables. Please be aware of this restriction: **When a run-time error terminates a program, some variable values printed by the Debugger may be inaccurate.**

Dynamic allocation incorrect

When a 2-byte block was allocated from a 4-byte free block in memory, the remaining 2 bytes in the free block were not handled correctly during execution. The problem caused memory fragmentation or even odd address traps when the program ran.

Multiple errors on file output

The error message mechanism attempts to write any partial output line to the output file as part of the `close` operation after a fatal error. If the error was detected during a write to an output file, such as a disk, the attempt triggered the "multiple errors detected" message instead of the actual problem. In V2.1B, the support library marks the second occurrence of an I/O error for file output as "non-fatal." This should prevent the program from aborting with the "multiple errors detected" message.

Miscellaneous

Incorrect results were obtained from functions that returned structured-type values.

Run-time traps and incorrect calculations occurred when operations were performed on packed structures.

Spurious compile-time errors were reported when conformant array parameters were used with nested procedures.

Changes to RSX

Version 2.1B for RSX corrected these problems:

'Getpos' returned incorrect location

The `getpos()` procedure did not always return the correct location of the next record about to be read when the input file was a `text` file.

'Put' didn't always put

When a terminal, line printer, or paper tape punch was opened as a file type other than `text`, data was not correctly written to the device when a `put()` was used. For

example, if a terminal was opened as a `file of integer`, and `put()` statements used to write data to the terminal (two characters per integer), nothing was printed on the terminal.

'Ioerror' always 'true'

If `ioerror` was the first operation performed on a **text** file, an interaction with Lazy I/O caused the return of the value **true** even when no error existed. Work-around: use the fourth parameter of the **reset** or **rewrite** call to determine whether the file is correctly opened. The value of the fourth parameter is -1 if the file cannot be opened.

Errors not trapped

The run-time errors "File is not a random access file" and "Seek() to record zero" were not correctly trapped when `noioerror()` was used with a file.

Profiler overlay omitted

The overlay description file PAS.ODL for V2.1A did not explain how to overlay a Pascal–2 program compiled with the /PROFILE option. For V2.1B, PAS.ODL properly describes overlay procedures for programs that use the Profiler.

'Not enough memory'

When a Pascal task is initialized and the section `$$HEAP` has not been expanded, the support library attempts to allocate 2K words for the stack by extending the task. This works fine until the size of the task reaches 30K words. At this point, less than 2K words remain for the stack and the compiler reports "not enough memory." In 2.1B, Pascal–2 uses any remaining space for the stack when a task exceeds 30K words, allowing users to write larger tasks before using overlays.

Incorrect file size returned

When the fourth parameter is used with the **reset** procedure, Pascal–2 should return the size of the file, in blocks. However, Pascal–2 was returning a file size of one block for an empty file. The file size for an empty file is now reported as zero.

/APD/RW causes lost line

A line of information may be lost under the following conditions: an existing **text** file is opened via a **reset** with the /APD/RW switches to permit appending data to the file, and a **write** statement is used without a **writeln** to write the characters to the file, and the file is then closed. The problem is caused by the use of **reset** to open the file; **reset** normally opens files for input only. When the /APD/RW switches are used, the file is really an output file, so any partial lines should be flushed out when the file is closed. Use **writeln** to be sure that the last line is written to the file before the file is closed.

'Sayerr' incorrect

The **sayerr** routine did not correctly print the text for I/O error when the RSX I/O error codes and directive error codes (stored in a byte) were ambiguous. In version 2.1B, the directive error codes are biased by 128 to distinguish them from I/O error codes. V2.1A users should be aware that errors such as "Illegal user buffer" might be reported when "Device driver not resident" is actually the error.

Record lengths for 'text' files

When **text** files containing lines longer than 132 characters were opened with a **reset** statement, a "record length error" could be generated because Pascal allocates a maximum buffer size of 132 characters. Work-around: use the I/O control switch `/VAR:`*nnn* where *nnn* is the maximum line length of the data in the file. The value *nnn* can be greater than 132.

'Rename' didn't

When an explicit version number was given for a file to be renamed and that file was not the most current version, the **rename** operation completed as expected, but the directory entry for the most current version of the file was removed instead of the entry for the file that was renamed. V2.1A users should not give explicit version numbers for files being renamed.

Character dropped on line wraps

When a line longer than 132 characters is written to a **text** file, the line wraps and the line is broken at 132 characters as if by a **writeln** statement. Under V2.1A, a character was dropped when the line is wrapped this way on disk files. Terminal output wraps correctly. Work-around: use a **writeln** statement before outputting more than 132 characters.

Message vague

The run-time error message "multiple errors detected" does not provide any information about the actual errors detected in the program. The error message is printed when a run-time error occurs while an error message is being printed, and it usually indicates a severe error, such as writing over support library code. In version 2.1B, the support library causes a run-time trap so that RSX prints the contents of the hardware registers as follows:

R0	User PC of original error
R1	Error code for original error
R2	Secondary error PC
R3	Secondary error code
R4	I/O status if I/O error

The error codes are in Appendix B: Run-Time Error Messages of the Programmer's Guide.

Errors, additions to manuals

Sample program errs

The sample program REVERSE listed in the *Pascal–2 User Manual* for V2.1 does not operate properly. The second line of the program being reversed is not printed because of an interaction with lazy I/O and the way the REVERSE program is written. To operate correctly, a boolean variable called **Done** must be inserted into the program. The main loop in the program must be modified as shown:

```
repeat
  new(x);
  with x^ do Getpos(f, block, offset);
  x^.next := p;
  p := x;
  Done := eof(f);
  if not Done then readln(f);
until Done;
```

Changes to RSTS

Version 2.1B for RSTS corrects these problems:

No record-oriented I/O under RSX

When the RSX version of Pascal is run under the RSX emulator on RSTS, it is not possible to perform I/O on record-oriented devices other than the user's terminal. The RSTS system returns an error status of −37 for all such I/O requests because the operating system does not properly simulate RSX I/O to record devices such as the line printer. I/O to the user's terminal is handled correctly. In V2.1B, code has been added to enable the support library to recognize RSTS emulation and perform I/O to record devices correctly.

'Read' may hang

A **read** performed on an integer or real hangs forever if the user types ^Z with I/O error trapping turned on and without entering a valid number first.

Second real value lost

The value of the second real variable is lost in programs that attempt to read two reals in succession, but only when the second real contains fewer characters than the first. Workaround: add a leading 0 to the second real.

Iostatus printed incorrect values

Iostatus produced the wrong values when an I/O error occurred, causing the error routine to print either a zero or a large negative number.

Modules missing in /eis version

The EIS version of the support library for V2.1A did not contain modules for the exponent function and the natural logarithm.

Miscellaneous

LIBDEF.PAS was not properly reflecting the data stored in the file variable. This resulted in FDUMP not giving proper results as well as prohibiting the user from testing any of the device status conditions accurately.

OPFDMP.PAS has been modified to properly print the device number if there is one.

OPERRO.PAS now prints the unit number of the device if an I/O error occurred.

The entry point for **P$DISP** was in the user library when it should have been in the run-time system. Attempts to run the examples in the manual that accessed this entry would fail.

Changes to RT-11

Error reporting improved

The **fatal** error status at location 53 octal was not being properly set or tested by the support library. This resulted in some command files not terminating when a fatal error was encountered.

In conjunction with the error status problem, the low-level routines (.READ/WRITE) would also set the error condition bit if a fatal transfer error occurred. This would cause Pascal to print the "multiple errors" message.

The **Sayerr** routine was added to print the error text of low-level RT-11 system calls.

Iostatus would produce the wrong values when an I/O error occurred, causing the error routine to print either a zero or a large negative number. **Iostatus** has been modified to return a negative value if a system error occurs on a file, a zero if no error occurs, and a positive number if the problem is a support library error. The error number is returned as a full 16-bit integer value.

The support library now saves the error condition code (if any) from the last **reset** or **rewrite** operation so that the user may call **Ioerror** and **Iostatus** to determine a possible cause if the operation fails. This will work properly only if called before any other I/O operation is performed.

The reason is that the saved location is also set by the .READ/WRITE code since **reset** could imply a .READ, which could cause the failure.

'Read' may hang on reals

A **read** performed on an integer or real hangs forever if the user types ^Z with I/O error trapping turned on and without entering a valid number first.

Second real value lost

The value of the second real variable is lost in programs that attempt to read two reals in succession, but only when the second real contains fewer characters than the first. Workaround: add a leading 0 to the second real.

Pascal has a standard

Pascal now officially has an international standard. The International Standards Organization, in a vote tallied this summer, adopted a standard language definition for Pascal. The action followed earlier adoption of an American standard that is a subset of the international one, lacking only conformant array parameters.

The sequence leading to adoption of the Pascal standard began in December 1982, when ANSI and IEEE agreed on the American standard, identical to the international draft standard except for conformant array parameters. At the same time, the Joint Pascal Committee of ANSI and IEEE recommended adoption of the international standard Level 1 (including conformant array parameters) to the U.S. committee known as X3J9, which then voted "yes" on the international standard at the next meeting. Previously, the U.S. was one of three "no" votes. This time, the ISO standard passed with no dissenting votes and one abstention.

Work continues on extensions

The Joint Pascal Committee of ANSI and IEEE continues to work on a "candidate extension library" for extensions that will resolve the known weaknesses of standard Pascal. The ANSI-IEEE committee has agreed on a number of minor issues but has not resolved differences over major issues. Proposals are circulating on these issues, which remain "work in progress." A report is due in December.

Copies of the international standard are available from:

Document number BS 6192:1982.
British Standards Institute
Marylands Avenue
Hemel Hempstead
Herts HP 2 4SQ

Information exchange

If you need information on technical applications involving Pascal, or if you have an application that might interest other users, send us a brief description for inclusion in the "Information Exchange." Your description should follow the format of the items below. Interested parties may contact one another directly.

Log Management System and RMS Interface, for RSX-11M; a menu-driven 50-program package that incorporates a custom-developed interface (available separately) between Digital's Record Management Services (RMS) and Pascal–2 V2.OK. The log package supports a multiplicity of data reporting features with subsystems for accounting, timber sales, and sales data accumulation. The RMS interface gives the user access to all of the RMS sequential, random, and multi-keyed capabilities. Contact: Jim Bombardier, Wasser and Winters Co., P.O. Box 396, Longview, WA 98632, (206) 423-1080.

DCL Command PASCAL, for the Oregon Software Pascal–2 V2.1A compiler; software package contains patches and files to enable a new DCL command **PASCAL** and to make **HELP PASCAL** work. All files are for RSX-11M V4.0 only. For information, contact: Mr. Bruce Williams, OPUS Coordinator, c/o EOCOM, 15771 Red Hill Avenue, Tustin, California 92680.

Distributors attend annual workshop

Oregon Software Distributors from Europe, Canada, Japan and Australia received technical information from Oregon Software development teams and exchanged viewpoints on the needs of customers at their second annual workshop.

The two days of discussions in Portland followed DEXPO East and preceded Oregon Software's sixth Annual Open House. The first day was devoted to technical presentations and a hands-on session, where distributors ran concurrent-programmed games on the 68000, used SourceTools in a simulated development environment, and compiled programs under UNIX on the PDP-11. The second day was devoted to round-table discussions in which participants exchanged views on the needs of customers, the quality and timeliness of written materials, and the need for additional information.

Customer contact is the key for sales staff

In the last issue, we introduced our marketing director, David Cloutier, and manager for general distributors, Pat Rau. We'd also like you to know our sales staff, who they are and what they do.

Lorie, Terry, and Tim handle customer support and sales. Their jobs require them to spend lots of time on the phone, providing product information, following up on sales, and responding to customers' requests for help. They respond to inquiries from prospective customers, which come in response to advertising, articles, or as referrals from our friends. The sales staff provides prospects and current customers alike with additional product information.

Their jobs don't end with a sale. They rectify any shipping problems and keep their accounts in support.

Lorie Griffith

Lorie, our senior sales person, has a strong science background, including some contact with computers as a chemistry major in college and as an assayist on her first job. Later, she became a computer operator for a sawblade manufacturer. Before she joined Oregon Software, Lorie did customer support for an applications software company. Her major project was a large automated inventory and accounting system for an automobile importer. As part of an 18-month contract, she did some programming and wrote user documentation in addition to providing support.

Lorie lives in rural Washington, where she raises livestock and is active in community projects.

Tim McMenamin

Tim says that "people make the job worthwhile"; he likes the service to customers as much as the software itself. He enjoys negotiating with distributors and finds the contact with large firms stimulating.

A native of Wisconsin, Tim came to Oregon for skiing and schooling. A journalism and political science major at college, he worked on the staff of a state senator and a congressman. Before coming to Oregon Software, Tim sold DBMS applications and CP/M program generators for a small software house. He likes the technical challenge of sales and is taking computer science courses to build his programming knowledge.

When he's not at work or school, Tim enjoys all forms of skiing, camps out in the local mountains, and runs for exercise.

Terry Juve

Terry is a recent addition to the staff. Trained as a programmer, she likes to work with other programmers. She also enjoys direct contact with the customer.

Terry recently earned an associate degree in applications programming at Portland Community College, and she's starting on a bachelor's degree in computer science at Portland State University.

Her sales background has been in commercial casuality insurance for various brokerage houses in Portland, and she's taken business courses as well.

Between work and school, she likes to spend the little bit of free time she has reading science fiction and psychology; she likes to cook, hike, dance, and play an occasional softball game.

Bug Contest winners crowned

After some tough decisions, we have the winners for the Bug Contest (see Newsletter No. 4). We couldn't decide on a "best" bug so we have co-winners. The prize category itself forced us to make some hard choices, but we finally decided on an official Oregon Software "Party Pascal" baseball hat, plus a bottle of Oregon wine with which the winners may begin a party at which they may wear their new hats. The winners are:

Best Bug	Herje Wikegaard (SERN Dator Konsult) Lee Mattiesen (NW Oklahoma State U.)
Application	Joann H. Schultz (Lockheed Electronics)
Best Prize	Doug Foster (E-systems)

This issue marks a transition

Oregon Software's seventh Pascal Newsletter marks a transition. David Spencer officially takes over as editor, and I step down. Actually, David has coordinated the writing and production of the last several issues. I have written a few articles and have assisted with the review and rewriting of others, but David has been doing the vast majority of the work.

David is also taking over as manager of the technical writing group. I am returning to the ranks of simply "writer." This changeover seems to be an appropriate time to review our company's writing efforts since July of 1980, when Oregon Software's first technical writer (yours truly) was hired.

We instituted the newsletter with the ultimate goal of quarterly issues. The one-man writing staff started slowly. After adding two staff members — David and Mike Kuhn, who has been our primary manual specialist — we have quickened the pace for a total of 7 in about 30 months. Time constraints still force us to combine issues on occasion, but the result has been the publication of newsletters with more substance. This one, for instance, is the combined Summer-Fall issue, with special emphasis on concurrent programming. It's the largest and most technically oriented issue this year.

We plan to continue with at least one staff-produced technical article and one user-produced article each issue. You should note that we run as many good user articles as we get, and we're always looking for more. Please write or call David with your ideas. We'll also continue with our regular fare of product announcements, OPUS columns, book reviews, letters, etc. We hope to have a more detailed Bug Log, including better lists of known bugs, both fixed and unfixed. Our biggest problem is that the production and mailing processes require several weeks' time, and the number of known bugs, and their status, can change dramatically after the newsletter has gone to press.

For our documentation in general, our goal has been to have manuals that reflect the high quality of our software. Further, we wanted a plain style that described complex software in the simplest terms possible. To a large degree, I believe we have succeeded. This summer marked the release of our new 2.1 manuals. Mike was the primary author of the new manuals, which not only document new software features but also provide much more detail on existing features. The considerable expansion of the 2.1 manuals represents about six months of Mike's hard work, with occasional help from the rest of the writers and a good deal of assistance (as always) from our programming staff.

In addition to manuals for new products, including SourceTools and several Pascal–2-based development systems, we are also working on new manuals for our original Pascal product, Pascal–1. A supplement to the existing Pascal–1 manuals is complete, and a revised manual similar to the Pascal–2 manuals should be done this fall. One of David's biggest tasks will be that of bringing all of our manuals up to the uniformly high standards of our Pascal–2 product line.

We need your help, by the way. Many of the improvements in our manuals have resulted from comments by users — a small but vocal group of users, as it happens. I'd like to take this last opportunity as outgoing editor to solicit comments and questions about our newsletter and documentation. Though our materials are reviewed thoroughly in-house before release, we can't really know whether they meet the needs of the users unless you tell us.

Collins Hemingway

Pascal
NEWSLETTER
OREGON SOFTWARE

2340 SW Canyon Road
Portland, Oregon 97201

David Spencer, editor
Collins Hemingway and Michael Kuhn, staff writers
Erin Ryan, production assistant

The Pascal Newsletter is published quarterly by Oregon Software, Inc., 2340 SW Canyon Rd., Portland, OR 97201; (503) 226-7760. Each customer of Oregon Software receives one free subscription per site. Additional subscriptions are available upon written request.

The Pascal Newsletter accepts articles of interest to Pascal users: solutions to troublesome programming situations, new applications of Pascal, interesting variations on standard applications, etc. Submit articles on paper (typed and double-spaced), on floppy disk, or on magnetic tape, sent to the attention of the editor. We pay $100 per newsletter page for any article we print.

Copyright © 1983 by Oregon Software, Inc.
ALL RIGHTS RESERVED.

RSTS, RSX, RT-11, PDP-11, VAX/VMS, and IAS are trademarks of Digital Equipment Corp. UNIX is a trademark of Western Electric. Pascal–1, Pascal–2, SourceTools and Pascal Newsletter are trademarks of Oregon Software.

Printed in USA